Journal of Instruction-Level Parallelism 5 (2003) 1-35 Submitted 10/02; published 04/03

Optimizing SMT Processors for High Single-Thread Performance

Gautham K.Dorai GAUTHAMT@GLUE.UMD.EDU
Donald Yeung YEUNG@ENG.UMD.EDU
Seungryul Choi SRCHOI@ENG.UMD.EDU
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies
University of Maryland at College Park

Abstract

Simultaneous Multithreading (SMT) processors achieve high processor throughput at the expense of single-thread performance. This paper investigates resource allocation policies for SMT processors that preserve, as much as possible, the single-thread performance of designated "foreground" threads, while still permitting other "background" threads to share resources. Since background threads on such an SMT machine have a near-zero performance impact on foreground threads, we refer to the background threads as *transparent threads*. Transparent threads are ideal for performing low-priority or non-critical computations, with applications in process scheduling, subordinate multithreading, and on-line performance monitoring.

To realize transparent threads, we propose three mechanisms for maintaining the transparency of background threads: slot prioritization, background thread instruction-window partitioning, and background thread flushing. In addition, we propose three mechanisms to boost background thread performance without sacrificing transparency: aggressive fetch partitioning, foreground thread instruction-window partitioning, and foreground thread flushing. We implement our mechanisms on a detailed simulator of an SMT processor, and evaluate them using 8 benchmarks, including 7 from the SPEC CPU2000 suite. Our results show when cache and branch predictor interference are factored out, background threads introduce less than 1% performance degradation on the foreground thread. Furthermore, maintaining the transparency of background threads reduces their throughput by only 23% relative to an equal priority scheme.

To demonstrate the usefulness of transparent threads, we study *Transparent Software Prefetching* (TSP), an implementation of software data prefetching using transparent threads. Due to its near-zero overhead, TSP enables prefetch instrumentation for all loads in a program, eliminating the need for profiling. TSP, without any profile information, achieves a 9.41% gain across 6 SPEC benchmarks, whereas conventional software prefetching guided by cache-miss profiles increases performance by only 2.47%.

1. Introduction

Simultaneous multithreading (SMT) processors achieve high processor throughput by exploiting ILP between independent threads as well as within a single thread. The increased processor throughput provided by SMT, however, comes at the expense of single-thread performance. Because multiple threads share hardware resources simultaneously, individual threads get fewer resources than they would have otherwise received had they been

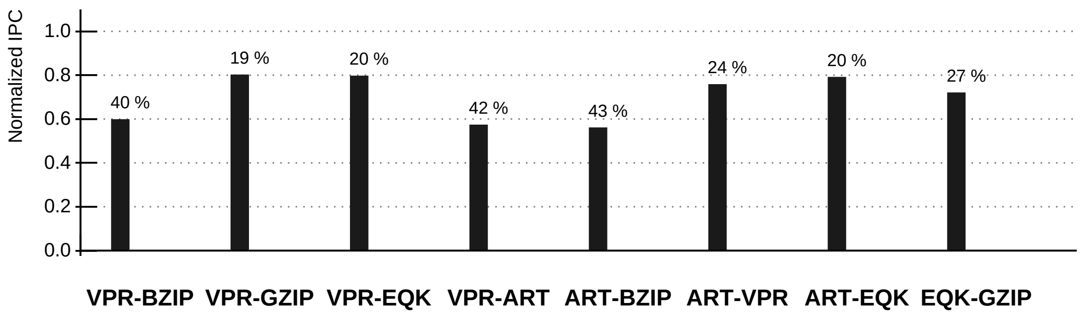

Figure 1: IPC of a single benchmark running in a two-benchmark multiprogrammed workload normalized to its single-threaded IPC on a dedicated SMT. Percentages appearing on top of the bars report reduction in single-thread performance due to simultaneous execution.

running alone. Furthermore, existing resource allocation policies, such as ICOUNT [1] and FPG [2], favor threads with high ILP, steering resources to threads whose instructions pass through the pipeline the most efficiently. Threads that utilize processor resources less efficiently receive fewer resources and run even slower.

To demonstrate this reduction in single-thread performance, Figure 1 shows the IPC of individual threads running in a multiprogrammed workload. Different bars in Figure 1 represent different multiprogrammed workloads, each consisting of two benchmarks running simultaneously on an 8-way SMT machine with a 128-entry instruction window depth. In these experiments, the SMT processor uses the ICOUNT policy as its fetch allocation mechanism. The benchmarks are selected from the SPEC CPU2000 benchmark suite, and the labels under each bar name the two benchmarks participating in each workload. The performance bars themselves report the IPC of one of the benchmarks (the left-most benchmark in each bar's label) normalized to that benchmark's single-threaded IPC on a dedicated processor, and the label atop each bar reports the reduction in single-thread performance. Figure 1 shows under the ICOUNT resource allocation policy, the performance of individual threads is reduced by roughly 30% across all the workloads.

Our work investigates resource allocation policies for SMT processors that preserve, as much as possible, the single-thread performance of designated high-priority or "foreground" threads, while still permitting other low-priority or "background" threads to share resources. Our approach allocates resources to foreground threads whenever they can use them and regardless of how inefficiently those resources might be used, thus permitting foreground threads to run as fast as they would have run on a dedicated SMT machine. At the same time, we only allocate *spare* resources to background threads that foreground threads would have otherwise left idle, thus allowing background threads to share resources without degrading foreground thread performance. Since the background threads on such an SMT machine are imperceptible to the foreground threads (at least from a resource sharing standpoint), we refer to the background threads as *transparent threads*.

Transparent threads are ideal for performing low-priority or non-critical computations. Several applications of multithreading involve such low-priority computations, and thus map naturally onto transparent threads:

Process Scheduling. Transparent threads can assist in scheduling multiprogrammed workloads onto SMT processors. When a latency-sensitive process enters a multiprogrammed workload (for example, when an interactive process receives an event), all non-latency-sensitive processes can be down-graded to run as transparent threads. During the time that the latency-sensitive or foreground process is active, the transparent threads yield all processor resources necessary for the foreground process to run as fast as possible. At the same time, the transparent threads are not shut out completely, receiving any resources that the foreground process is unable to use.

Subordinate Multithreading. Transparent threads can support subordinate multithreading [3, 4]. Subordinate threads perform computations on behalf of a primary thread to increase its performance. Recently, there have been several proposals for subordinate multithreading, using subordinate threads to perform prefetching (also known as *pre-execution*) [5, 6, 7, 8, 9, 10], cache management [11], and branch prediction [3]. Unfortunately, the benefit of these techniques must be weighed against their cost. If the overhead of subordinate computation outweighs the optimization benefit, then applying the optimization may reduce rather than increase performance. For this reason, detailed profiling is necessary to determine when optimizations are profitable so that optimizations can be applied selectively to minimize overhead.

Transparent threads enable subordinate multithreading optimizations to be applied *all the time*. Since transparent threads never take resources away from foreground threads, subordinate threads that run as transparent threads incur zero overhead; hence, optimizations are always profitable, at worst providing zero gain. With transparent threading support, a programmer (or compiler) could apply subordinate threading optimizations blindly. Not only does this relieve the programmer from having to perform profiling, but it also increases the optimization coverage resulting in potentially higher performance.

Performance Monitoring. Finally, transparent threads can execute profiling code. Profiling systems often instrument profile code directly into the application [12, 13, 14]. Unfortunately, this can result in significant slowdown for the host application. To minimize the impact of the instrumentation code, it may be possible to perform the profiling functionality inside transparent threads. Similar to subordinate multithreading, profile-based transparent threads would not impact foreground thread performance, and for this reason, could enable the use of profiling code all the time.

This paper investigates the mechanisms necessary to realize transparent threads for SMT processors. We identify the hardware resources inside a processor that are critical for single-thread performance, and propose techniques to enable background threads to share them transparently with foreground threads. In this paper, we study the transparent sharing of two resources that impact performance the most: *instruction slots* and *instruction buffers*. We also discuss transparently sharing a third resource, *memories*, but we do not evaluate these solutions in this paper. Next, we propose techniques to boost transparent thread performance. Using our basic resource sharing mechanisms, transparent threads receive hardware resources only when they are idle. Under these conservative assumptions, transparent threads can exhibit poor performance. We propose additional techniques that detect when resources held by foreground threads are not critical to their performance, and aggressively reallocate them to transparent threads.

To study the effectiveness of our techniques, we undertake an experimental evaluation of transparent threads on a detailed SMT simulator. Our evaluation proceeds in two parts. First, we study transparent threads in the context of multiprogramming. These experiments stress our mechanisms using diverse workloads, and reveal the most important mechanisms for enforcing transparency across a wide range of applications. We find our mechanisms are quite effective, permitting low-priority processes running as transparent threads to induce less than 1% performance degradation on high-priority processes (excluding cache and branch predictor interference). Furthermore, our mechanisms degrade the performance of transparent threads by only 23% relative to an equal priority scheme. Second, we study *Transparent Software Prefetching* (TSP), an implementation of software data prefetching of affine and indexed array references [15] using transparent threads. By off-loading prefetch code onto transparent threads, we achieve virtually zero-overhead software prefetching. We show this enables software prefetching for all candidate memory references, thus eliminating the need to perform profiling a priori to identify cache-missing memory references. Our results show TSP without profile information achieves a 9.41% gain across 6 SPEC benchmarks, whereas conventional software prefetching guided by cache-miss profiles increases performance by only 2.47%.

The rest of the paper is organized as follows. Section 2 presents our mechanisms in detail. Next, Section 3 describes our simulation framework used to perform the experimental evaluation. Then, Section 4 studies transparent threads in the context of multiprogramming and Section 5 studies TSP. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2. Transparent Threads

This section presents the mechanisms necessary to support transparent threads. First, Section 2.1 discusses the impact of resource sharing on single-thread performance in an SMT processor. Next, Section 2.2 presents the mechanisms for transparently sharing two classes of resources, instruction slots and buffers, and discusses possible solutions for transparently sharing a third class, memories. Finally, Section 2.3 presents the mechanisms for boosting transparent thread performance.

2.1 Resource Sharing

Figure 2 illustrates the hardware resources in an SMT processor pipeline. The pipeline consists of three major components: fetch hardware (multiple program counters, a fetch unit, a branch predictor, and an I-cache), issue hardware (instruction decode, register rename, instruction issue queues, and issue logic), and execute hardware (register files, functional units, a D-cache, and a reorder buffer). Among these hardware resources, three are dedicated. Each context has its own program counter and return stack (the return stack is part of the branch predictor module in Figure 2). In addition, each context effectively has its own register file as well since the integer and floating point register files, while centralized, are large enough to hold the architected registers from all contexts simultaneously. All other hardware resources are shared between contexts.

Simultaneously executing threads increase processor throughput by keeping shared hardware resources utilized as often as possible, but degrade each others' performance by com-

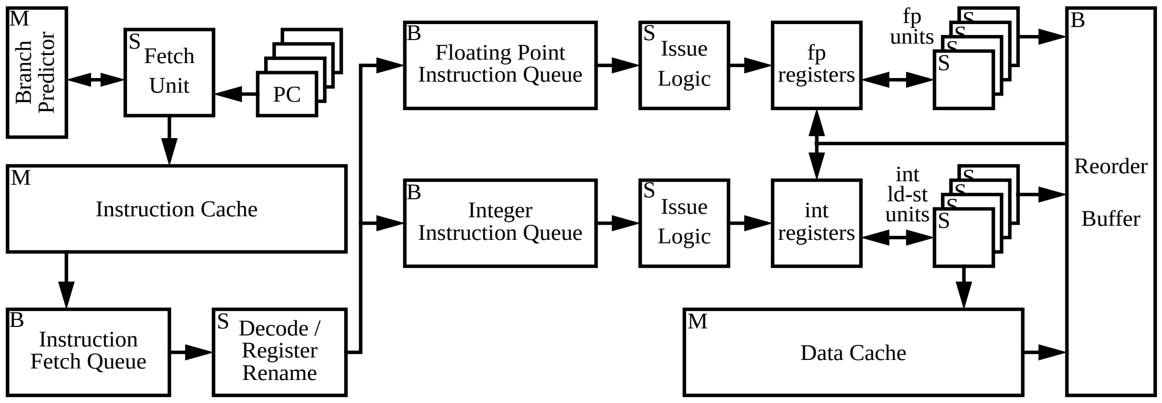

Figure 2: SMT processor hardware (diagram adapted from [1]). Each shared hardware resource is labeled with a letter signifying one of three resource classes: instruction slots (S), instruction buffers (B), and memories (M).

peting for these resources. The goal of transparent threading, therefore, is to allocate a shared resource to a background thread only when it is not competing with a foreground thread for the same resource. To provide more insight, we group the shared hardware resources into three classes–instruction slots, instruction buffers, and memories–and discuss their resource allocation properties. Each shared resource in Figure 2 is labeled with its resource class, using the labels “S,” “B,” and “M,” respectively.

Instruction slots are pipeline stages. The fetch, decode, rename, issue, writeback, and commit stages contain instruction slots, typically equal in number to the width of the machine. In addition, functional units also contain slots, typically one per functional unit per cycle of latency (assuming a fully pipelined unit). Instruction buffers hold stalled instructions. Figure 2 shows four buffers: the instruction fetch queue holds fetched instructions waiting to be decoded and renamed, the integer and floating point instruction queues hold instructions waiting on operands and/or functional units, and the reorder buffer holds instructions waiting to commit. Finally, memories are cache structures. The I- and D-caches as well as the branch predictor tables in Figure 2 make up this category.

The ramifications for allocating a shared resource to a background thread depend on its resource class. Allocating an instruction slot to a background thread impacts the foreground thread on the *current cycle only*. Instructions normally occupy slots for a single cycle. While there are exceptions to this rule (for example a load instruction that suffers a cache miss in its data-cache access stage), we find these cases do not create resource conflicts frequently. Therefore, background threads can use instruction slots transparently as long as there is no conflict with the foreground thread on the cycle of allocation. In contrast, allocating an instruction buffer entry to a background thread potentially impacts the foreground thread on *future cycles*. Instructions typically occupy buffers for many cycles, particularly in the reorder buffer where instructions remain until all preceding instructions (including those performing long-latency operations) commit. Therefore, allocating a buffer entry to a background thread can cause resource conflicts with the foreground thread in the future even if the resource is idle on the cycle of allocation.

Compared to instruction slots and buffers, memory resource sharing has a less direct impact on foreground thread performance. Rather than taking an execution resource away from the foreground thread, the use of memory resources by a background thread can increase the number of performance-degrading events experienced by the foreground thread (*i.e.*, branch mispredictions and cache misses). Similar to instruction buffers, the impact does not occur at the time of use, but rather, at a point in the future.[1]

2.2 Transparency Mechanisms

Having discussed the resource sharing problem in Section 2.1, we now present several mechanisms that permit background threads to share resources transparently with the foreground thread. We present one mechanism for sharing instruction slots, two for sharing instruction buffers, and finally, we discuss possible solutions for sharing memories.

Instruction Slots: Slot Prioritization. Since instruction slots are normally held for a single cycle only, we allocate an instruction slot to a background thread as long as the foreground thread does not require the slot on the same cycle. If the foreground thread competes for the same instruction slot resource, we give priority to the foreground thread and retry the allocation for the background thread on the following cycle. We call this mechanism *slot prioritization.*

As described in Section 2.1, every pipeline stage has instruction slots; however, we implement slot prioritization in the fetch and issue stages only. We find that prioritizing slots in additional pipeline stages does not increase the transparency of background threads. To implement slot prioritization in the fetch stage, we modify the SMT processor's fetch priority scheme. Our default scheme is ICOUNT [1]. When choosing the threads to fetch from on each cycle, we artificially increase the instruction count for all background threads by the total number of instruction window entries, thus giving fetch priority to foreground threads always regardless of their instruction count values. Background threads receive fetch slots only when the foreground thread cannot fetch, for example due to a previous I-cache miss or when recovering from a branch misprediction. Slot prioritization in the issue stage is implemented in a similar fashion. We always issue foreground thread instructions first; background thread instructions are considered for issue only when issue slots remain after all ready foreground thread instructions have been issued.

Instruction Buffers: Background Thread Instruction-Window Partitioning. Compared to instruction slots, transparently allocating instruction buffer resources is more challenging because resource allocation decisions impact the foreground thread on future cycles. It is impossible to guarantee at allocation time that allocating an instruction buffer entry to a background thread will not cause a resource conflict with the foreground thread. Determining this would require knowing for how long the background thread will occupy the entry as well as knowing the number of buffer entries the foreground thread will request in the future.

We propose two solutions for transparently allocating instruction buffers. The first solution is the *background thread instruction-window partitioning* mechanism, illustrated in

1. One shared resource left out of our discussion here is rename registers. From our experience, there is very little contention on rename registers given a reasonable number of them. Hence, we do not consider rename register sharing in our design of transparent threads.

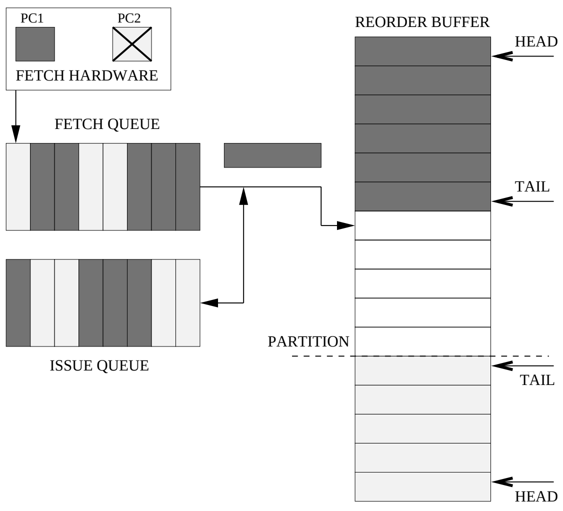

Figure 3: Background thread instruction-window partitioning mechanism. Dark shaded blocks denote foreground thread instructions, while light shaded blocks denote background thread instructions.

Figure 3. In this mechanism, we limit the maximum ICOUNT value for the background threads. When a background thread reaches this instruction count limit, its fetch stage is locked to prevent it from fetching further and consuming additional instruction buffer entries. The background thread remains locked out of the fetch stage until its ICOUNT value drops. Effectively, this approach imposes a hard partition on the buffer resources and never permits the background thread to overstep this partition, as shown in Figure 3.

The background thread instruction-window partitioning scheme ensures the total number of background thread instructions in the instruction fetch queue and the reorder buffer never exceeds its instruction count limit. Notice this does not guarantee that background threads never take instruction buffer resources away from the foreground thread. If the foreground thread tries to consume most or all of the instruction buffer resources, it can still "collide" with the background threads in the buffers and be denied buffer resources. However, this scheme limits the damage that background threads can inflict on the foreground thread. By limiting the maximum number of buffer entries allocated to background threads, a large number of entries can be reserved for the foreground thread. Another drawback of this mechanism is that it limits the instruction window utilization for the background thread, and therefore degrades background thread performance. As Figure 3 illustrates, even when additional "free" reorder buffer entries are available, the background thread cannot make use of them if it has reached its ICOUNT limit due to the hard partition.

Instruction Buffers: Background Thread Flushing. In our second scheme for transparently allocating instruction buffers, we permit background threads to occupy as many instruction buffer entries as they can (under the constraint that the foreground thread gets all the fetch slots it requests), but we pre-emptively reclaim buffer entries occupied by background threads when necessary. We call this mechanism *background thread flushing*.

Background thread flushing works in the following manner. First, we trigger background thread flushing whenever the foreground thread tries to allocate an instruction buffer entry

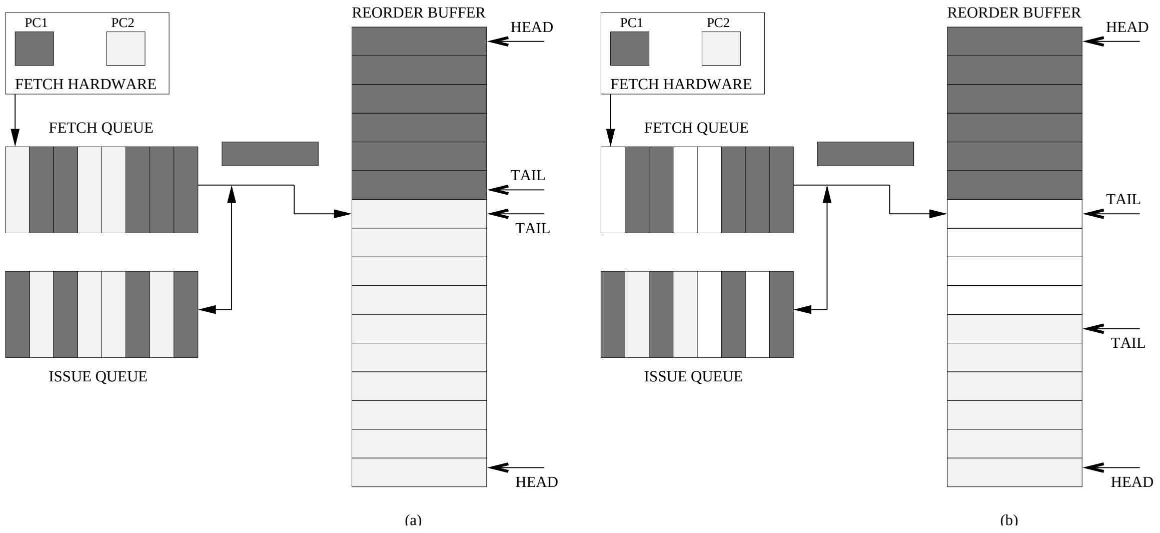

Figure 4: Background thread flushing mechanism - (a) collision between background and foreground threads in the reorder buffer triggers flushing, and (b) state of the buffers after flushing. Dark shaded blocks denote foreground thread instructions, while light shaded blocks denote background thread instructions.

but all entries of that type are filled. There are four instruction buffers, as shown in Figure 2, whose allocation can trigger flushing: the instruction fetch queue, the integer and floating point instruction queues, and the reorder buffer. Among these four instruction buffers, we have observed that reorder buffer contention is responsible for the most performance degradation in the foreground thread (in fact, contention for the other instruction buffers usually occurs when the reorder buffer is full). For simplicity, we trigger flushing only when the foreground thread is unable to allocate a reorder buffer entry. Figure 4(a) illustrates such a scenario where the foreground and background threads have collided in the reorder buffer, blocking the entry of a new foreground thread instruction. At this point, we trigger flushing.

Once flushing is triggered, we select a background thread to flush. We compare the ICOUNT values of all background threads and pick the thread with the largest value. From this thread, we flush the N youngest instructions in the reorder buffer, where N is the width of the machine. (If the selected background thread occupies fewer than N reorder buffer entries, we flush all of its entries). Figure 4(b) illustrates the state of the buffers after flushing has been applied to the example in Figure 4(a). In addition to flushing the reorder buffer, we also flush any instructions in the integer or floating point instruction queues corresponding to flushed reorder buffer entries, and we flush all instruction fetch queue entries belonging to this thread as well. Finally, we roll back the thread's program counter and register file map to the youngest unflushed instruction. Notice our flushing mechanism is similar to branch misprediction recovery; therefore, most of the hardware necessary to implement it already exists. However, our mechanism requires checkpointing the register file map more frequently since we flush to an arbitrary point in the reorder buffer rather than to the last mispredicted branch. In Section 3, we will discuss techniques for reducing the cost of implementing background thread flushing.

Compared to background thread instruction-window partitioning, background thread flushing requires more hardware support; however, it potentially permits background threads to share instruction buffer resources more transparently. Background thread flushing guarantees the foreground thread always gets instruction buffer resources, using pre-emption to reclaim resources from background threads if necessary. At the same time, background thread flushing can provide higher throughput to background threads compared to background thread instruction-window partitioning. If the foreground thread does not use a significant number of instruction buffer entries, the background threads can freely allocate them because there is no limit on the maximum number of entries that background threads can hold.

Memories: Possible Solutions. As our results in Section 4 will show, sharing instruction slot and instruction buffer resources has the greatest impact on foreground thread performance, while sharing memories has a less significant performance impact. For this reason, we focus on the first two classes of resources, and we do not evaluate mechanisms for transparently sharing memories in this paper.

We believe memory resources, *e.g.*, branch predictor tables and caches, can be transparently shared using approaches similar to those described above. One possible approach is to limit the maximum number of locations that a background thread can allocate in the memories. Memory resources are used by mapping an address to a memory location. For branch predictors, a combination of the branch address and a branch history pattern is typically used to index into the branch predictor table. For caches, a portion of the effective memory address is used to index into the cache. Consequently, utilization of the memory resources can be limited by modifying the mapping function and using a reduced number of address bits to form the index. Background threads can use the modified mapping function, hence using a fewer number of memory locations. Foreground threads can use the normal mapping function to access the full resources provided by the memories.

2.3 Performance Mechanisms

In Section 2.2, we focused on maintaining background thread transparency; however, achieving high background thread performance is also important. Unfortunately, as Section 4 will show, the resource sharing mechanisms presented in Section 2.2 can starve background threads, leading to poor performance. This section presents several additional mechanisms for increasing resource allocation to background threads without sacrificing transparency. We present one mechanism for increasing fetch slot allocation, and two mechanisms for increasing instruction buffer allocation.

Fetch Instruction Slots: Fetch Partitioning. The most important instruction slot resources are the fetch slots because the frequency with which a thread receives fetch slots determines its maximum throughput. As described in Section 2.2, fetch slot prioritization always gives priority to the foreground thread by artificially increasing the ICOUNT values of the background threads. Even though the foreground thread always gets priority for fetch, the background thread can still get a significant number of fetch slots if the SMT employs an aggressive *fetch partitioning scheme.*

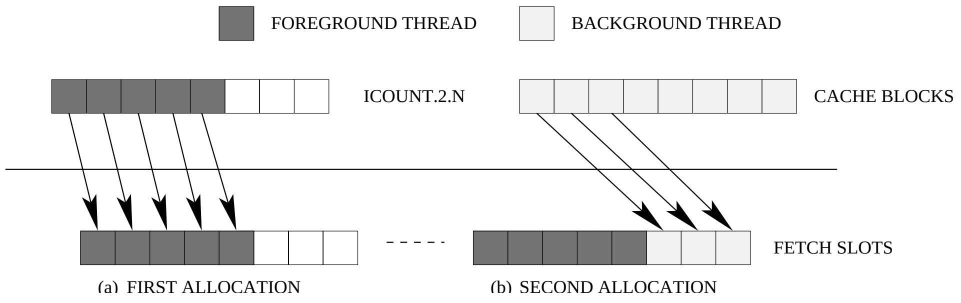

Figure 5: Slot Prioritization with the ICOUNT.2.N fetch partitioning mechanism. (a) First, the foreground thread is allocated all the fetch slots it needs. (b) Then, the background thread is allocated the remaining fetch slots.

The most basic fetch partitioning scheme is to permit only one thread to fetch every cycle, and to give all the fetch slots to that single thread. Assuming an ICOUNT fetch priority scheme, this basic fetch partitioning scheme is called ICOUNT.1.N [1], where N is the fetch width. Under ICOUNT.1.N with slot prioritization, background threads receive fetch slots only when the foreground thread cannot fetch at all. If the foreground thread fetches even a single instruction, all N fetch slots on that cycle are allocated to the foreground thread since only one thread can fetch per cycle. In our SMT processor model, we assume the only times the foreground thread cannot fetch are 1) if it has suffered an I-cache miss, in which case it stalls until the cache miss is serviced, or 2) if it has suffered a branch mispredict, in which case it stalls until mispredict recovery completes.

If instead of allowing only a single thread to fetch per cycle, multiple threads are allowed to fetch per cycle, then background threads can receive significantly more fetch slots. In this paper, we evaluate the ICOUNT.2.N [1] fetch partitioning scheme which chooses up to N instructions for fetch from 2 threads every cycle. Under ICOUNT.2.N with slot prioritization, the foreground thread still gets highest priority for fetch; however, background threads can fetch anytime the foreground thread is unable to consume all N fetch slots on a given cycle. Consider the slot allocation sequence illustrated in Figure 5. The ICOUNT.2.N fetch partitioning scheme fetches one cache block each from the foreground thread and the background thread. The slot prioritization transparency mechanism first allocates all the fetch slots needed to the foreground thread, as shown in Figure 5(a). Under the ICOUNT.1.N scheme, the background thread would not be able to get any fetch slots on the cycle depicted in Figure 5(a). However, with the ICOUNT.2.N fetch partitioning mechanism, any of the unused fetch slots will be allocated to the background thread, as shown in Figure 5(b).

In our SMT processor model, we assume the foreground thread terminates fetching on a given cycle if it encounters a predict-taken branch or if it fetches up to an I-cache block boundary. Under these assumptions, it is rare for the foreground thread to fetch N instructions per cycle, opening up significantly more spare slots for background threads to consume.

Instruction Buffers: Foreground Thread Instruction-Window Partitioning. The combination of mechanisms described in Section 2.2 can easily starve background threads of instruction buffer resources. Since the foreground thread always gets fetch priority under slot prioritization, and since the background thread's allocation of instruction buffer entries is limited under either background thread instruction-window partitioning or background thread flushing, it is possible for the foreground thread to consume all instruction buffer resources. Once this happens, the background thread may rarely get buffer entries even if it is allocated fetch slots.

We propose two solutions for increasing background thread instruction buffer allocation that mirror the mechanisms for transparently allocating instruction buffers presented in Section 2.2. First, just as we limit the maximum ICOUNT value for background threads, we can also limit the maximum ICOUNT value for foreground threads. When the foreground thread reaches this instruction count limit, it is not allowed to consume additional fetch slots until its ICOUNT value drops. We call this mechanism *foreground thread instruction-window partitioning*.

By limiting the maximum number of foreground thread instructions in the instruction buffers, we reserve some buffer entries for the background threads. However, similar to background thread instruction-window partitioning, this approach is not completely transparent since it allows background threads to take resources away from the foreground thread. The performance impact can be minimized, though, by choosing a large foreground thread ICOUNT limit.

Instruction Buffers: Foreground Thread Flushing. Our results in Section 4 will show foreground thread instruction window partitioning improves the throughput of the background thread at the expense of its transparency. Foreground thread flushing is a more dynamic mechanism that allows the background thread to reclaim instruction buffer entries from the foreground thread. Unfortunately, arbitrary flushing of the foreground thread, just like instruction window partitioning, will degrade the performance of the foreground thread. However, if triggering of foreground thread flushing is carefully chosen, it is possible to reclaim instruction buffer entries from the foreground thread without impacting its performance.

We trigger foreground thread flushing when a cache-missing load from the foreground thread reaches the head of the reorder buffer. Our experiments show applications frequently fill up the reorder buffer whenever there is a load miss at its head. When a cache-missing load is at the head of the reorder buffer, the rest of the instruction buffer entries are stagnant (because of in-order commit) and the foreground thread cannot proceed until the load instruction at the head of the reorder buffer completes. Hence, flushing these stagnant instructions will not impact the foreground thread performance as long as they are restored before the cache-missing load completes. Once the foreground thread instructions have been flushed, we temporarily prevent the foreground thread from fetching and reclaim the flushed entries. After a certain number of cycles, which is determined by the number of cycles left before the load at the reorder buffer head completes, the foreground thread is allowed to commence fetching again.

Figure 6 presents a pictorial representation of a foreground thread flush sequence. In Figure 6(a), the reorder buffer is full with foreground thread instructions which are stagnant due to a load miss at the head of the reorder buffer. New background thread instructions cannot enter the reorder buffer because of this condition. Figure 6(b) shows our mechanism triggers a flush of a certain number of stagnated foreground thread instructions from the tail,

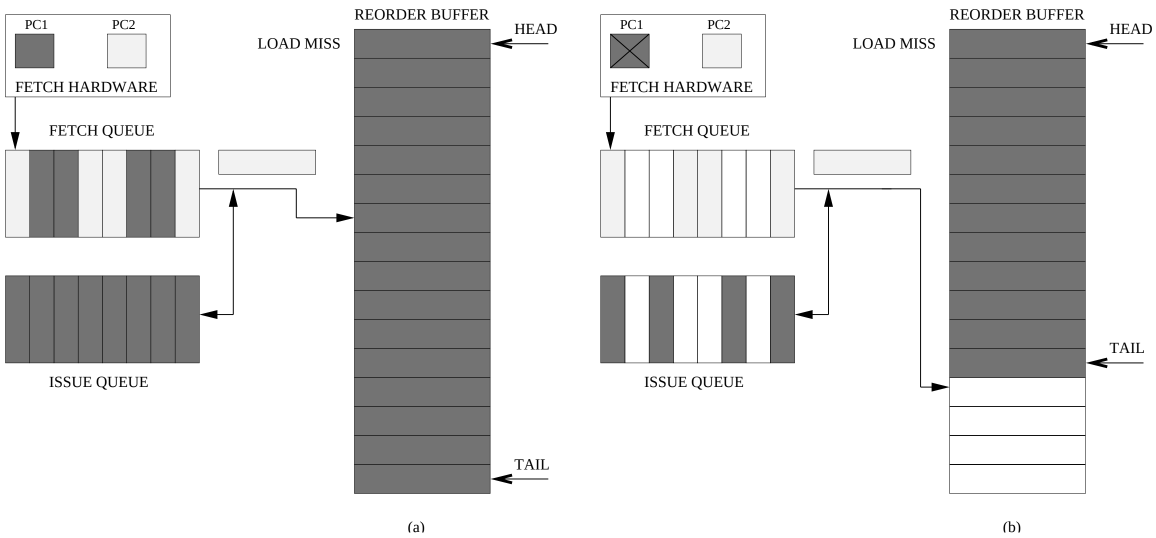

Figure 6: Foreground thread flushing mechanism - (a) a reorder buffer full of foreground thread instructions and with a cache-missing load at its head triggers a flush, and (b) state of the buffers after flushing. Dark shaded blocks denote foreground thread instructions, while light shaded blocks denote background thread instructions.

and locks out the foreground thread from fetching. Similar to background thread flushing, we flush the youngest foreground thread instructions from the tail of the reorder buffer, all corresponding instructions in the integer and floating point instruction queues, and all instructions in the instruction fetch queue belonging to the foreground thread. Now, the background thread can make forward progress using these freed up buffer entries. After a certain number of cycles, the foreground thread is allowed to fetch again, thereby reclaiming the flushed out entries.

Residual Cache Miss Latency. To avoid degrading foreground thread performance, the number of flushed instructions, F, and the number of cycles we allow for flush recovery, T, must be commensurate with the number of cycles that the cache-missing load remains stalled at the head of the reorder buffer. We call this time the *residual cache-miss latency*, R. If R is large, we can afford to flush more foreground thread instructions since there is more time for recovery, thus freeing a larger number of buffer entries. However, if R is small, we must limit the number of flushed instructions since the recovery time is itself limited. Because we expect R to vary on every cache miss, we rely on hardware to estimate R each time we initiate foreground thread flushing, and then select appropriate F and T values to dynamically control the number of flushed instructions and the timing of flush recovery. In Section 3, we will discuss the choice of F and T values as a function of R. One important issue, though, is how do we determine R?

We use a cycle counter for every foreground thread load instruction that suffers a cache miss to estimate R. When a load instruction initially suffers a cache miss, we allocate a cycle counter to the load, clear the counter contents, and increment the counter on every cycle thereafter. When the cache-missing load reaches the head of the reorder buffer and stalls, we use the following heuristic to estimate R. We assume a cache-missing load that reaches the head of the reorder buffer is either a) an L1 miss that hits in the L2 cache if it has been in the reorder buffer for less than the L2 hit time, or b) an L2 miss if it has been in the reorder buffer for more than the L2 hit time. The residual cache-miss latency is then calculated by subtracting the counter value from the L2 hit time (when (a) is true) or the main memory latency (when (b) is true).

Overestimating the residual cache-miss latency can degrade the foreground thread performance because it will lead to overly aggressive flushing. We conducted experiments to compare the residual cache-miss latencies estimated using the above heuristic against the actual values measured by the simulator. Our experiments show our heuristic predicts the residual cache-miss latencies accurately in a majority of our applications. When a discrepancy exists between the estimated values and the measured values, our heuristic tends to underestimate rather than overestimate residual cache-miss latency.

3. Simulation Framework

Our simulation framework is based on the SMT simulator from [16]. This simulator uses the out-of-order processor model from SimpleScalar v2.0, augmented to simulate an SMT pipeline. Unless otherwise specified, our experiments use the baseline simulator settings reported in Table 1. These settings model a 4x8-way issue SMT processor with 32-entry integer and floating point instruction queues and a 128-entry reorder buffer. We also model a hybrid gshare/bimodal branch predictor. As indicated in Table 1, we assume a 3-cycle penalty after detecting a branch mispredict to recover from the wrong path of execution. In Sections 4.2 and 4.3, we initially assume other threads can fetch during this wrong-path recovery delay. Then, at the end of Section 4.3, we study a conservative wrong-path recovery policy that removes this assumption. Finally, we model a memory hierarchy consisting of a split 32K L1 cache and a unified 512K L2 cache. The caches and main memory are simulated faithfully; however, we do not model the queues between different memory hierarchy levels (*e.g.*, processor store buffers, write buffers, etc.). For our applications, we observe only modest memory concurrency; hence, we believe small queues would be sufficient to prevent stalls caused by these structures.

To evaluate transparent threads, we extended our basic SMT simulator to model the mechanisms presented in Section 2, namely the two mechanisms for sharing instruction slots (slot prioritization and fetch partitioning) and the four mechanisms for sharing instruction buffers (background and foreground thread instruction window partitioning, and background and foreground thread flushing). When simulating our instruction window partitioning schemes, we assume a maximum background and foreground thread ICOUNT limit of 32 and 112 instructions, respectively. For fetch partitioning, our simulator models both the ICOUNT.1.8 and ICOUNT.2.8 schemes, as discussed in Section 2.3 (our simulator also models an ICOUNT.1.16 scheme–we will describe this later in Section 4.3). ICOUNT.2.8 requires fetching 16 instructions from 2 threads (8 from each thread) on every cycle [1], and using slot prioritization to select 8 instructions out of the 16 fetched instructions. To provide the fetch bandwidth necessary, our I-cache model contains 8 interleaved banks, and accounts for all bank conflicts. In addition to simulating contention for I-cache banks, we also simulate contention for rename registers. We assume all contexts share 100 integer and

Processor Parameters			
Hardware Contexts	4	Issue Width	8
Fetch Queue Size	32 entries	Instruction Queue Size	32 Int, 32 FP entries
Load-Store Queue Size	64 entries	Reorder Buffer Size	128 entries
Int/FP Units	8/8	Int Latency	1 cycle
FP Add/Mult/Div Latency	2/4/12 cycles	Rename Registers	100 Int, 100 FP
Branch Predictor Parameters			
Branch Predictor	Hybrid gshare/Bimodal	gshare/Bimodal Size	4096/2048 entries
Meta Table Size	1024 entries	BTB Size	2048 entries
Return-Address-Stack Size	8 entries	Mispredict Recovery	3 cycles
Memory Parameters			
L1 Cache Size	32K I and 32K D (split)	L2 Cache Size	512K (unified)
L1/L2 Block Size	32/64 bytes	L1/L2 Associativity	4-way/4-way
L1/L2 Hit Time	1/10 cycles	Memory Access Time	122 cycles

Table 1: Baseline SMT simulator settings used for most of the experiments.

$R < 8$	F=0	T=0
$8 \leq R < 16$	F=8	T=4
$16 \leq R < 32$	F=16	T=8
$32 \leq R$	F=48	T=16

Table 2: Choice of the number of instructions to flush, F, and the number of flush recovery cycles, T, as a function of the residual cache-miss latency, R.

100 floating point rename registers in addition to the per-context architected registers, as indicated in Table 1.

As described in Section 2.3, our foreground thread flushing mechanism dynamically selects the number of instructions to flush, F, and the number of flush recovery cycles, T, based on the residual cache-miss latency, R, at the time flushing is triggered. Table 2 reports the F and T values used by our simulator for a range of R values. Since our flushing mechanisms (for both background and foreground threads) flush to an arbitrary point in the reorder buffer, they require frequent register map checkpointing (see Section 2.2). For maximum flexibility, checkpointing every instruction would be necessary. To reduce hardware cost, however, our simulator models checkpointing every 8th instruction only. When flushing is triggered, we compute the number of instructions to flush as normal, described in Sections 2.2 and 2.3 for background and foreground thread flushing, respectively. Then, we flush to the nearest checkpointed instruction, rounding up when flushing the background thread (more aggressive) and rounding down when flushing the foreground thread (more conservative).

In addition to the hardware specified in Tables 1 and 2, our simulator also provides ISA support for multithreading. We assume support for a `fork` instruction that sets the program counter of a remote context and then activates the context. We also assume support for `suspend` and `resume` instructions. Both instructions execute in 1 cycle; however, `suspend` causes a pipeline flush of all instructions belonging to the suspended context. Finally, we assume support for a `kill` instruction that terminates the thread running in a specified context ID. Our multithreading ISA support is used extensively for performing Transparent Software Prefetching, described later in Section 5.

Name	Type	Input	FastFwd	Sim
VPR	SPECint 2000	reference	60M	233M
BZIP	SPECint 2000	reference	22M	126M
GZIP	SPECint 2000	reference	170M	140M
EQUAKE	SPECfp 2000	reference	18M	1186M
ART	SPECfp 2000	reference	20M	71M
GAP	SPECint 2000	reference	105M	157M
AMMP	SPECfp 2000	reference	110M	2439M
IRREG	PDE Solver	144K nodes	29M	977M

Table 3: Benchmark summary. The first three columns report the name, type, and inputs for each benchmark. The last two columns report the number of instructions in the fast forward and simulated regions.

To drive our simulation study, we use the 8 benchmarks listed in Table 3. Four of these benchmarks are SPECInt CPU2000 benchmarks, three are SPECfp CPU2000 benchmarks, and the last is an iterative PDE solver for computational fluid dynamics problems. In all our experiments, we use functional simulation to fast forward past each benchmark's initialization code before turning on detailed simulation. The size of the fast forward and simulated regions are reported in the last two columns of Table 3.

4. Evaluating Transparent Threads

Our experimental evaluation of transparent threads consists of two major parts. First, in this section, we characterize the performance of our transparent threading mechanisms. Then, in Section 5, we investigate using transparent threads to perform software data prefetching.

4.1 Methodology

This section characterizes the performance of our transparent threading mechanisms by studying them in the context of multiprogramming. We perform several multiprogramming experiments, each consisting of 2 - 4 benchmarks running simultaneously on our SMT simulator. A single benchmark from the workload is selected to run as a foreground thread, while all other benchmarks run as background threads. From these experiments, we observe the degree to which our mechanisms maintain background thread transparency (Section 4.2) as well as the ability of our mechanisms to increase transparent thread throughput (Section 4.3).

From the 8 applications listed in Table 3, we use the first 5 for our multiprogramming experiments, grouping benchmarks together based on resource usage characteristics. Of particular significance is a benchmark's *reorder buffer occupancy*. Benchmarks with high reorder buffer occupancy (typically caused by frequent long-latency cache misses) use more instruction buffer resources, whereas benchmarks with low reorder buffer occupancy use fewer instruction buffer resources. Among our 5 benchmarks, BZIP and ART have high occupancy, EQUAKE and GZIP have low occupancy, while VPR has medium occupancy. In order to stress our mechanisms and to study their behavior under diverse workload

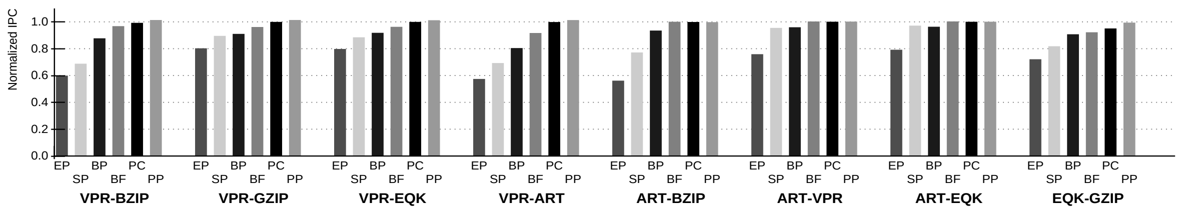

Figure 7: Normalized IPC of the foreground thread when running simultaneously with a single background thread. The bars represent different transparent sharing mechanisms: equal priority (EP), slot prioritization (SP), background thread instruction window partitioning (BP), background thread flushing (BF), private caches (PC), and private predictors (PP).

characteristics, we group together benchmarks that exhibit both high and low reorder buffer occupancy, using both types as foreground and background threads.

4.2 Background Thread Transparency

Figures 7 and 8 report the normalized IPC of the foreground thread when running simultaneously with a single background thread and with three background threads, respectively. Groups of bars represent sets of simultaneously running benchmarks, each specified with a label that names the foreground benchmark first followed by the background benchmark(s). Bars within each group represent different transparent sharing mechanisms from Section 2.2 applied incrementally. In particular, the first four bars report normalized IPC with no mechanisms (*i.e.*, all threads have equal priority), with slot prioritization, with background thread instruction window partitioning and slot prioritization, and with background thread flushing and slot prioritization, labeled EP, SP, BP, and BF, respectively. All experiments use the ICOUNT.2.8 fetch partitioning scheme, with all other background thread performance mechanisms disabled. Finally, all bars are normalized to the IPC of the foreground thread running on a dedicated SMT machine (*i.e.*, without any background threads).

Figure 7 shows background thread flushing with slot prioritization (BF bars) is the most effective combination of transparent sharing mechanisms. With these mechanisms, the foreground thread achieves 97% of its single-thread performance averaged across the 8 benchmark pairs, compared to only 70% of single-thread performance when pairs of benchmarks are run with equal priority (EP bars). Background thread instruction window partitioning with slot prioritization (BP bars) also provides good transparency, with the foreground thread achieving 91% of its single-thread performance; however, our results show BP is less effective than BF in all cases. Slot prioritization alone (SP bars) is the least effective, allowing the foreground thread to achieve only 84% of its single-thread performance. Figure 8 shows the same qualitative results as Figure 7, demonstrating our mechanisms are just as effective when maintaining transparency for multiple background threads.

Having quantified the transparency of our background threads, we now examine the extent to which the foreground thread's performance degradation is due to sharing memories, a type of resource sharing that our mechanisms do not address. In our SMT model,

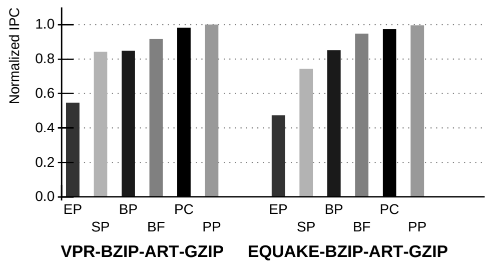

Figure 8: Normalized IPC of the foreground thread when running simultaneously with three background threads. The bars are the same as those in Figure 7.

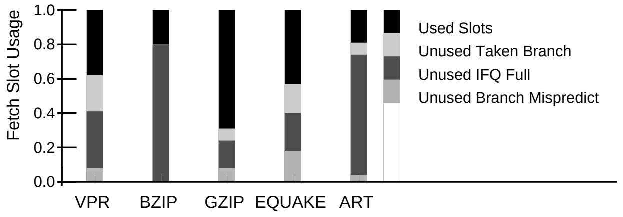

Figure 9: Fetch slot usage for our benchmarks when each benchmark is run on a dedicated SMT processor.

threads share two types of memory structures: caches and branch predictor tables. To isolate the impact of sharing these structures on foreground thread performance, we replicate them, thus removing any contention due to sharing. The last two bars of each group in Figures 7 and 8 report the normalized foreground thread IPC assuming the best mechanisms (*i.e.*, those used for the BF bars) when each context has private L1 and L2 caches (PC bars), and when each context has both private caches and a private branch predictor (PP bars). These results show when cache and branch predictor conflicts are removed, the foreground thread achieves essentially all of its single-thread performance. We conclude that our mechanisms enable the background threads to use instruction slots and instruction buffers in a completely transparent fashion, and that further improvements in foreground thread performance can only come by addressing memory sharing.

While Figures 7 and 8 quantify the extent to which background threads are transparent, they do not provide insight into how our mechanisms achieve transparency. To address this issue, we first study how our benchmarks use processor resources. Figure 9 illustrates the usage of the fetch stage, a critical SMT resource. In Figure 9, we break down the total available fetch slots into used and unused slots when each benchmark is run on a dedicated SMT processor. Unused slots are further broken down into three categories indicating the cause for the unused slots: wasted slots around a taken branch (after the branch on the same cycle and before the target on the next cycle), a full instruction fetch queue, and recovery from a branch mispredict. (A fourth possible category is I-cache stalls, but an insignificant number of unused slots are due to I-cache stalls in our benchmarks, so we omit this category in Figure 9).

Figure 9 sheds light on why our transparent threading mechanisms work. First, the "IFQ Full" components indicate the degree to which our benchmarks occupy instruction buffers, showing that BZIP and ART have high instruction buffer occupancy. In Figure 7, we see that any workload using these benchmarks as a background thread exhibits poor foreground thread performance under equal priority. When using equal priority, BZIP and ART frequently compete for instruction buffer entries with the foreground thread, degrading its performance. Consequently, in these workloads, background thread flushing significantly improves foreground thread performance since flushing reclaims buffer entries, making the foreground thread resilient to background threads with high instruction buffer occupancy. Conversely, Figure 9 shows GZIP and EQUAKE have low instruction buffer occupancy. In Figure 7, we see that any workload using these benchmarks as a background thread exhibits reasonable foreground thread performance under equal priority, and only modest gains due to flushing.

Second, anytime a workload uses a benchmark with a large "IFQ Full" component as a foreground thread, slot prioritization provides a large foreground thread performance gain and background thread flushing becomes less important. In Figure 7, the ART-VPR and ART-EQK (and to some extent, ART-BZIP) workloads exhibit this effect. When slot prioritization is turned on, ART gets all the fetch slots it requests and thus acquires a large number of instruction buffer entries (due to its high instruction buffer occupancy), resulting in a large performance boost. At the same time, the background thread receives fewer buffer entries, reducing the performance impact of flushing.

4.3 Transparent Thread Performance

Figure 10 reports the normalized IPC of the background thread using background thread flushing and slot prioritization for the multiprogrammed workloads from Figure 7. Bars within each workload group represent different transparent thread performance mechanisms from Section 2.3 applied incrementally. Specifically, we report normalized IPC with the ICOUNT.1.8 fetch partitioning scheme without and with foreground thread flushing, with the ICOUNT.2.8 fetch partitioning scheme without and with foreground thread flushing, with the ICOUNT.2.8 scheme and foreground thread instruction window partitioning, and with no mechanisms (*i.e.*, equal priority), labeled 1B, 1F, 2B, 2F, 2P, and EP, respectively. All bars are normalized to the IPC of the background thread running on a dedicated SMT machine.

Comparing the 4 left-most group of bars in Figure 10 reveals two results. First, we see the ICOUNT.1.8 fetch partitioning scheme provides the lowest background thread performance, allowing the background thread to achieve only 19% of its single-thread performance on average. Moreover, going from ICOUNT.1.8 (1B and 1F bars) to ICOUNT.2.8 (2B and 2F bars) provides a significant increase in background thread IPC. This is particularly true in workloads where the foreground thread exhibits a large number of "Taken Branch" unused fetch slots (*e.g.*, VPR and EQUAKE as shown in Figure 9) since this is the resource that ICOUNT.2.8 exploits compared to ICOUNT.1.8. Second, Figure 10 also shows foreground thread flushing is important across all workloads, for both ICOUNT.1.8 (1F bars) and ICOUNT.2.8 (2F bars). With foreground thread flushing, the background thread achieves 38% and 46% of its single-thread performance using the ICOUNT.1.8 and ICOUNT.2.8

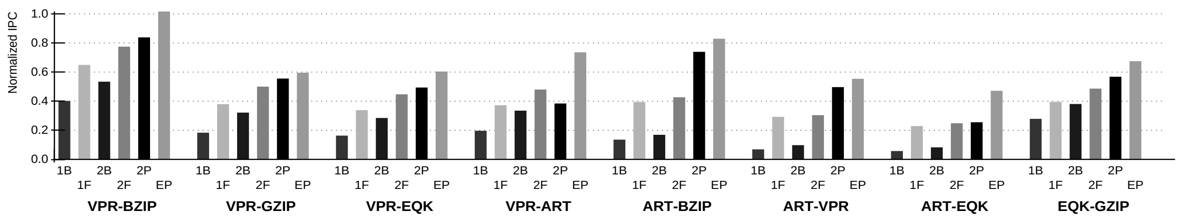

Figure 10: Normalized IPC of the background thread when the foreground thread runs simultaneously with a single background thread. The bars represent different transparent thread throughput mechanisms: ICOUNT.1.8 without (1B) and with (1F) foreground thread flushing, ICOUNT.2.8 without (2B) and with (2F) foreground thread flushing, ICOUNT.2.8 with foreground thread window partitioning (2P), and equal priority (EP). All bars use background thread flushing with slot prioritization.

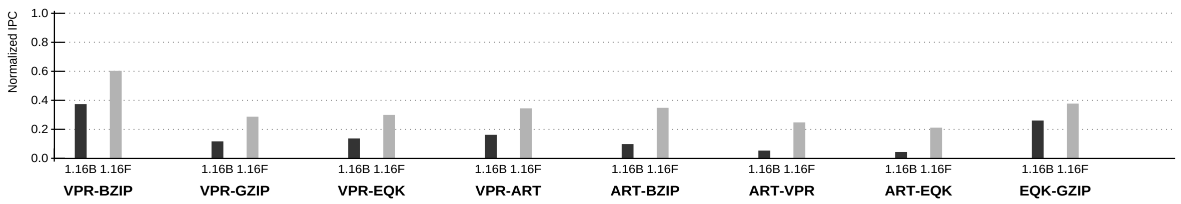

Figure 11: Normalized IPC of the background thread when the foreground thread runs simultaneously with a single background thread. The bars represent the ICOUNT.1.16 fetch configuration without (1.16B) and with (1.16F) foreground thread flushing. All bars use background thread flushing with slot prioritization.

schemes, respectively. Furthermore, our results show flushing is more important when the foreground thread has a high instruction buffer occupancy (*e.g.*, ART as shown in Figure 9). In these workloads, foreground thread flushing can provide the background thread with significantly more instruction buffer resources, resulting in large performance gains. Overall, foreground thread flushing combined with ICOUNT.2.8 (2F bars) improves the IPC of the background thread to within 23% of the equal priority scheme (EP bars).

Compared to ICOUNT.1.8, ICOUNT.2.8 offers two benefits. For one, it exploits a higher I-cache bandwidth. While both schemes fetch up to 8 instructions per cycle, ICOUNT.2.8 is more likely to achieve this maximum fetch throughput because it performs 2 I-cache accesses to pull up to 16 instructions from the I-cache (8 for each thread) per cycle, while ICOUNT.1.8 performs only 1 I-cache access per cycle. Another benefit is ICOUNT.2.8 provides more flexibility during fetch since it fetches from two threads every cycle rather than just one. To provide insight into the individual importance of these two benefits, Figure 11 studies another fetch scheme, ICOUNT.1.16. Like ICOUNT.1.8, ICOUNT.1.16 fetches from only one thread per cycle; however, like ICOUNT.2.8, it performs 2 I-cache accesses per cycle, providing up to 16 fetched instructions and permitting fetch past 1 taken branch every cycle. In Figure 11, we again see the importance of foreground thread flushing in boosting the throughput of the background thread. Overall, ICOUNT.1.16 with foreground thread flushing (1.16F bars) allows the background thread to achieve 36% of its

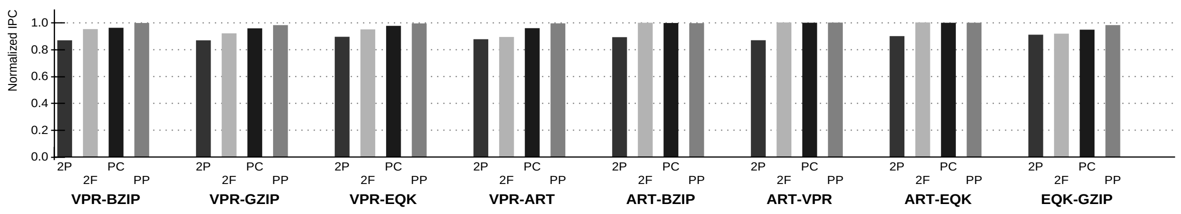

Figure 12: Normalized IPC of the foreground thread when running simultaneously with a single background thread. The bars represent different transparent thread throughput mechanisms: foreground thread instruction window partitioning (2P), foreground thread flushing (2F), private caches (PC), and private predictors (PP). All bars use background thread flushing with slot prioritization.

single-thread performance on average, which is 20% better than no flushing (1.16B bars).[2] However, comparing Figures 10 and 11, we see ICOUNT.2.8 still outperforms ICOUNT.1.16, allowing the background thread to achieve a larger fraction of its single-thread performance in all cases. This demonstrates that increasing fetch throughput alone cannot match the performance of ICOUNT.2.8; instead, fetching from multiple threads per cycle provides noticeable gains in background thread performance.

Returning to Figure 10, we see that foreground thread window partitioning combined with ICOUNT.2.8 (2P bars) achieves the highest background thread performance, allowing the background thread to achieve 56% of its single-thread performance. Overall, we see instruction window partitioning combined with ICOUNT.2.8 improves the IPC of the background thread to within 13% of the equal priority scheme (EP bars). Although foreground thread window partitioning provides the highest background thread throughput, we must also consider its impact on foreground thread performance. Figure 12 plots the normalized IPC of the foreground thread for several of the experiments in Figure 10. This data shows that the increased background thread performance of foreground thread instruction window partitioning compared to foreground thread flushing comes at the expense of reduced foreground thread performance (the 2F bars achieve 95% of single-thread performance whereas the 2P bars achieve only 84%). We conclude that foreground thread flushing is more desirable since it increases background thread performance without sacrificing transparency. Similar to Figures 7 and 8, the last two bars of Figure 12 remove cache and branch predictor conflicts from the 2F bars, showing that practically all of the remaining foreground thread performance degradation is due to memory sharing.

Instantaneous Behavior. Thus far, we have studied the average overall throughput of the background thread only; we now investigate instantaneous throughput to provide insight into the burstiness of the background thread's progress. In Figure 13, we show histograms of the fraction of executed instructions belonging to the background thread (y-axis) over time (x-axis) from two different workloads under the ICOUNT.2.8 scheme with

2. In this section, we only present background thread performance results for ICOUNT.1.16. We also examined its background thread transparency, and found similar results to the ICOUNT.2.8 scheme studied in Section 4.2. Overall the foreground thread achieves 97.5% of its single-thread performance under the ICOUNT.1.16 scheme. We omit the detailed results to conserve space.

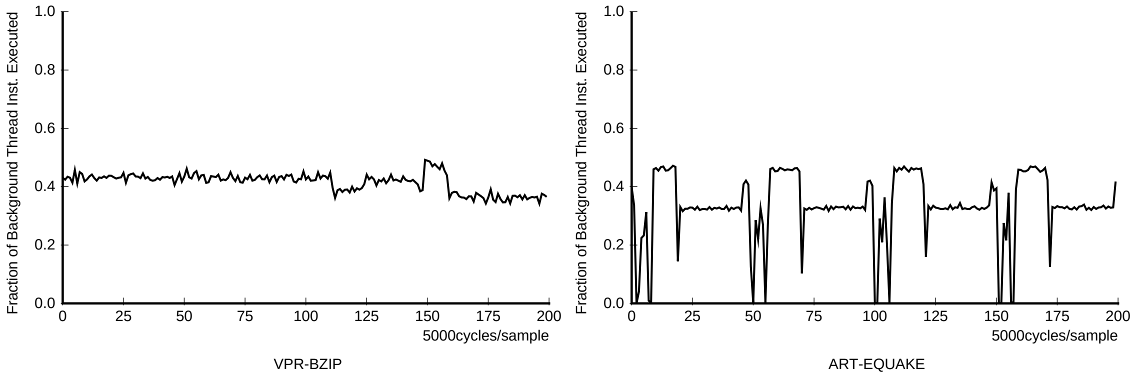

Figure 13: Histogram of the fraction of executed instructions from the background thread when running simultaneously with a single background thread. This fraction of executed instructions from the background thread is averaged over an interval of 5000 cycles.

foreground and background thread flushing. The workloads chosen are representative of the different behaviors we observed, as described below. Each datapoint in these time-based histograms reports a fraction averaged across a 5000-cycle period. In total, Figure 13 shows 200 datapoints (equivalent to 1 million cycles of execution) per workload.

Our results show that in workloads where the foreground thread has a low/medium instruction buffer occupancy (*e.g.*, VPR-BZIP, VPR-GZIP, VPR-EQK, VPR-ART, and EQK-GZIP), the histogram of executed background thread instructions is smooth across time, hovering around the 40% mark. However, in workloads where the foreground thread has a high instruction buffer occupancy (*e.g.*, ART-BZIP, ART-VPR, and ART-EQK), the histogram is spiky, with gaps where the background thread gets a very small fraction of execution resources. Usually, the gaps last for around 10,000 cycles. These gaps tend to form during intervals where the foreground thread fills up the instruction buffers but continues to make progress (*i.e.*, the foreground thread is not stalled on a long-latency memory operation). Since the foreground thread does not stall during these intervals, our flushing mechanisms are not triggered. This leads to gaps where a very low fraction of instructions are executed from the background thread.

Conservative Branch Mispredict Recovery. All our results presented so far assume an optimistic branch mispredict recovery policy: when a thread detects a mispredicted branch, we allow other threads to fetch during recovery from the wrong path of execution (which takes 3 cycles in our processor). In some machines, this assumption may not hold. For example, part of the recovery delay may involve informing the fetch stage that a particular thread has suffered a branch mispredict. In this case, the thread may continue to fetch useless wrong-path instructions after a branch mispredict has occurred, and compete for fetch slots during its mispredict recovery. By assuming a thread stops fetching immediately on a branch mispredict, we are fetching more non-speculative instructions than would potentially be possible in some machines. Fortunately, this optimistic use of fetch bandwidth does not impact the background thread transparency results presented in Section 4.2. Slot prioritization gives the foreground thread fetch priority over the background thread regardless of whether the background thread is recovering from a branch mispredict

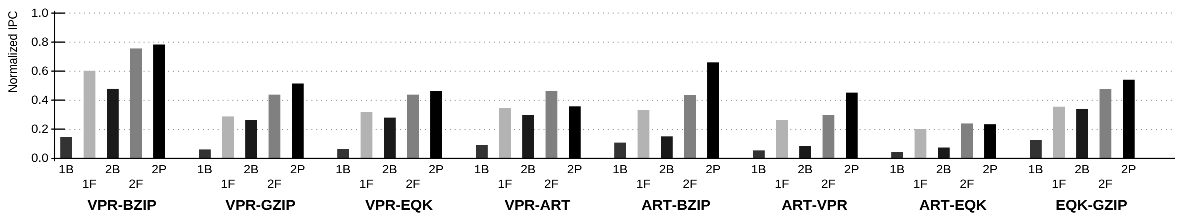

Figure 14: Normalized IPC of the background thread when the foreground thread runs simultaneously with a single background thread, and assuming a conservative branch mispredict recovery policy. All bars correspond to the bars presented in Figure 10.

or not. However, the background thread throughput results from Figures 10 and 11 may be overly optimistic since we are providing the background thread with fetch slots anytime the foreground thread is recovering from a branch mispredict.

In Figure 14, we report background thread throughput for the same experiments in Figure 10, but assuming a conservative branch mispredict recovery policy: the background thread is not allowed to fetch any instructions during branch mispredict recovery of the foreground thread. Essentially, this makes all the "Unused Branch Mispredict" slots in Figure 9 unavailable to the background thread. Notice this policy is overly conservative. In some cases, the background thread may still be able to fetch during foreground thread mispredict recovery, particularly if we employ mechanisms to improve background thread throughput. However, our intent is to show the worst-case impact of our optimistic assumptions.

The conservative recovery policy has a large impact on the "1B" bars. Compared to Figure 10, the background thread throughput goes down by 41.8% for these bars in Figure 14. Without any mechanisms to enhance background thread fetch, the "Unused Branch Mispredict" slots are the primary source of fetch bandwidth for the background thread; hence, removing them has a devastating effect on performance. However, as mechanisms are applied, background thread throughput becomes more resilient. For the "2B" bars, we see a smaller 12.8% reduction of the background thread throughput in Figure 14 compared to Figure 10. ICOUNT.2.8 permits the background thread to exploit "Unused Taken Branch" slots, reducing the impact of the lost "Unused Branch Mispredict" slots. For the "1F" and "2F" bars, the background thread throughput is only 9.8% and 1.9% lower, respectively, when comparing Figures 14 and 10. This is because foreground thread flushing permits the background thread to also exploit "Unused IFQ Full" slots. Finally, like foreground thread flushing, foreground thread window partitioning also opens up "Unused IFQ Full" slots for the background thread. Hence, the "2P" bars in Figure 14 show a modest 6.3% reduction in background thread throughput compared to Figure 10. Overall, we find our results do not change qualitatively when using a conservative branch mispredict recovery policy, but since some cases are impacted significantly, we use this policy in our remaining experiments.[3]

3. We do not report "EP" bars in Figure 14. The conservative branch mispredict recovery policy would lower the performance under "EP" as well. However, this would make our results look better, not worse, since our transparency and throughput mechanisms would compare more favorably against a lower "EP." Moreover, our branch mispredict recovery policy is overly conservative, so it would make EP look worse (and thus our results look better) than it really would be in a real machine. Hence, in our remaining results, we use the optimistic branch mispredict recovery policy for the "EP" bars.

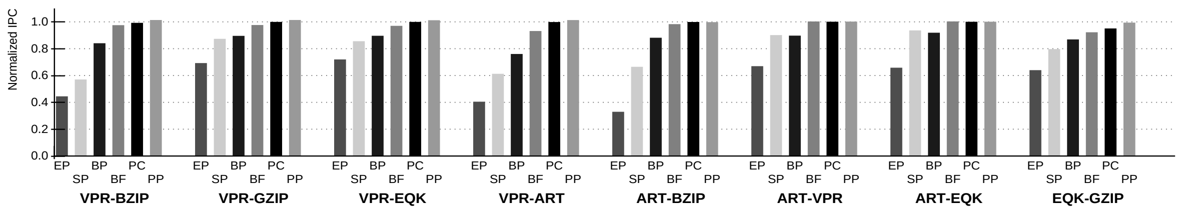

Figure 15: Normalized IPC of the foreground thread when running simultaneously with a single background thread on a 4-way machine with a 64-entry reorder buffer. The bars represent different transparent sharing mechanisms: equal priority (EP), slot prioritization (SP), background thread instruction window partitioning (BP), background thread flushing (BF), private caches (PC), and private predictors (PP).

4.4 Sensitivity Study

Sections 4.2 and 4.3 evaluate our mechanisms assuming the baseline SMT processor specified in Table 1. In this section, we study the sensitivity of our results to variations in the simulation parameters by exploring the performance of our mechanisms on a less aggressive processor. Specifically, we re-run the experiments from Figures 7 and 10 on a 4-way SMT processor with 16-entry integer and floating point instruction queues, and a 64-entry reorder buffer. We also reduce the fetch queue size to 16 instruction entries. All other simulator settings for the sensitivity experiments remain the same as those reported in Table 1.

Transparency mechanisms. First, we study the performance of the foreground thread on the reduced processor as transparency mechanisms are applied incrementally without any throughput mechanisms (*i.e.*, re-running the experiments from Figure 7). Figure 15 reports these experiments. Our first conclusion from Figure 15 is that even in a processor with limited resources, the combination of background thread flushing with slot prioritization (BF bars) still succeeds in maintaining the transparency of the background thread, allowing the foreground thread to achieve 98% of its single-thread performance. However, compared to the baseline results from Figure 7, the transparency mechanisms are stressed differently under the resource-constrained processor.

The equal priority scheme (EP bars) allows the foreground thread to achieve only 57% of its single-thread performance because of the increased contention for the limited processor resources. Slot prioritization is able to boost the foreground thread performance to only 77% of its single-thread performance. The remaining improvement in single-thread performance comes from the background thread flushing mechanism. The effectiveness of the background thread instruction window partitioning technique is also reduced, with the foreground thread achieving only 87% of its single-thread performance. From these results, we conclude that the background thread flushing mechanism becomes even more important for maintaining single-thread performance of the foreground thread on less aggressive processors.

Performance mechanisms. Next, we study the performance of the background thread on the reduced processor as the performance mechanisms are applied (*i.e.*, re-running the experiments from Figure 10). Figure 16 reports these experiments. As in the case of the foreground thread, the increased contention for processor resources on the reduced proces-

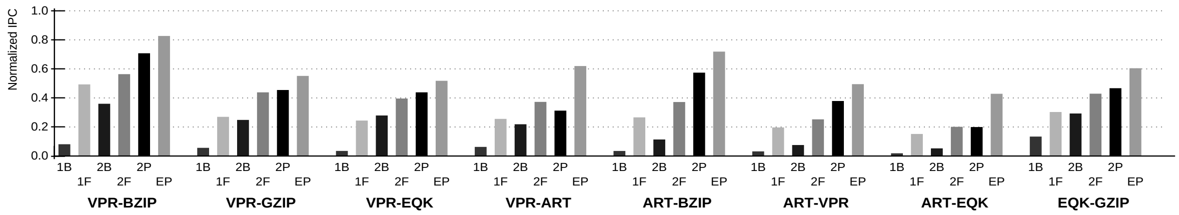

Figure 16: Normalized IPC of the background thread when the foreground thread runs simultaneously with a single background thread on a 4-way machine with a 64-entry reorder buffer. The bars represent different transparent thread throughput mechanisms: ICOUNT.1.8 without (1B) and with (1F) foreground thread flushing, ICOUNT.2.8 without (2B) and with (2F) foreground thread flushing, ICOUNT.2.8 with foreground thread window partitioning (2P), and equal priority (EP). All bars use background thread flushing with slot prioritization.

sor leads to lower single-thread performance of the background thread. The equal priority scheme (EP bars) allows the background thread to achieve only 60% of its single-thread performance. Similar to the results from Figure 10, Figure 16 shows moving from the ICOUNT.1.8 fetch partitioning scheme (1B bars) to the ICOUNT.2.8 scheme (2B bars) on the 4-way SMT processor significantly increases background thread performance. Furthermore, the foreground thread flushing mechanism (2F bars) boosts the background thread throughput to within 22% of the equal priority scheme, showing that this mechanism is still effective for the reduced processor. Finally, Figure 16 shows the foreground thread partitioning mechanism (2P bars) achieves the best performance, resulting in background thread throughput that is within 15% of the equal priority scheme. However, as was seen for the baseline processor, this mechanism is not transparent and degrades foreground thread performance on the reduced processor as well (these results have been omitted to conserve space).

5. Transparent Software Prefetching: Design and Evaluation

Software prefetching [17, 18, 15] is a promising technique to mitigate the memory latency bottleneck. It hides memory latency by scheduling non-blocking loads (*special prefetch instructions*) early relative to when their results are consumed. While these techniques provide visible latency-hiding benefits, they also incur limiting runtime overheads.

This section proposes and evaluates a new subordinate multithreading technique, called *Transparent Software Prefetching* (TSP). TSP performs software data prefetching by instrumenting the prefetch code in a separate *prefetch thread* rather than inlining it into the main computation code, as is done in conventional software prefetching. Prefetch threads run as background threads, prefetching on behalf of the computation thread which runs as a foreground thread. Because they run transparently, prefetch threads incur near-zero overhead, and thus almost never degrade the computation thread's performance.

TSP solves a classic problem associated with software prefetching: determining what to prefetch. Since conventional software prefetching incurs runtime overhead, it is important to instrument prefetching only for load instructions that suffer a sufficiently large mem-

(a) ORIGINAL LOOP

```
1  for (i = 0; i < N; i += 8) {
2    z = z + b[i];
3  }
```

(b) TRANSFORMED LOOP

```
1   /* Prologue Loop */
2   for (i = 0; i < PD; i += 8) {
3     prefetch(&b[i]);
4   }
5   /* Main Loop */
6   for (i = 0; i < N−PD; i+=8) {
7     prefetch(&b[i+PD]);
8     b = z + b[i];
9   }
10  /* Epilogue Loop */
11  for (; i < N; i += 8) {
12    z = z + b[i];
13  }
```

Figure 17: Conventional software prefetching example. (a) Original loop code. (b) Transformed loop with software prefetching.

ory access latency so that the benefit of prefetching outweighs the cost of executing the instrumentation code. Identifying the loads for which prefetching is profitable typically requires gathering detailed cache-miss profiles (*e.g.*, summary [19] or correlation [20] profiles). Unfortunately, such profiles are cumbersome to acquire, and may not accurately reflect memory behavior for arbitrary program inputs. In contrast, *TSP eliminates the need for profiling.* Since transparent sharing mechanisms guarantee prefetch threads never degrade the computation thread's performance, prefetching becomes profitable for *all* loads, regardless of their cache-miss behavior. Consequently, TSP can be applied naively, without ever worrying about the profitability of a transformation.

5.1 Implementation

Conventional Software Prefetching. As mentioned before, software prefetching schedules prefetch instructions to bring data into the cache before they are consumed by the program. Conventional software prefetching inlines the prefetch code along with the main computation code [17, 18, 15]. The inlined prefetches are software pipelined via a loop peeling transformation.

Figure 17(a) shows a simple code example, and Figure 17(b) illustrates the prefetch inlining and loop peeling transformations needed to instrument software prefetching for this code example. The transformations are designed to issue prefetches PD iterations in advance of the computation, also known as the *prefetch distance.* First, a prologue loop (lines 1-4) is inserted to issue the first PD prefetches without executing any computation code. Then, the main loop (lines 5-9) executes the computation code for all iterations except for the last PD iterations. This loop also prefetches data PD iterations ahead. Finally, the epilogue loop (lines 10-13) executes the last PD iterations of the loop without issuing any prefetches. The prologue, main, and epilogue loops perform the startup, steady-state, and draining phases of the software pipeline, respectively.

The software instrumentation shown in Figure 17 contributes runtime overhead to program execution. If the benefit due to prefetching does not outweigh the overhead due to software instrumentation, the overall execution time of the program degrades. Hence, it is important to instrument prefetching only when its benefit offsets its overhead. Prefetching is profitable for loads that exhibit high cache-miss ratios. To identify such loads, we acquire summary cache-miss profiles [19] to measure the miss ratio of individual loads, and evaluate the profitability of instrumenting prefetching for each load based on the following predicate:

$$PrefetchOverhead \quad < \quad L1_{miss\_rate} * L2_{hit\_time} + L2_{miss\_rate} * Mem_{latency} \qquad (1)$$

In computing the prefetch overhead, we assume that the instruction count overhead to prefetch each load is 12 instructions (8 instructions for the prologue loop + 1 prefetch instruction in the prologue loop + 2 instructions for the prefetch predicate in the main loop + 1 prefetch instruction in the main loop), and the average IPC for the loop is 1.5. Hence, the prefetch overhead is assumed to be 8 (12/1.5) cycles.

Transparent Software Prefetching. Instrumenting TSP involves several steps. First, we select any loop containing one or more affine array or indexed array references as a candidate for prefetch instrumentation. When nested loops are encountered, we consider prefetch instrumentation for the inner-most loop only. (Figure 18a shows an inner-most loop which we will use as an illustrative example). For each selected loop, we copy the loop header and place it in a separate *prefetch procedure* (Figure 18b, line 8). Inside the copied loop, we insert prefetch statements for each affine array and indexed array reference appearing in the original loop body (Figure 18b, lines 9-11).

Second, we insert code into the computation thread to initiate the prefetch thread (Figure 18a, lines 1-4). Since this code is executed by the computation thread, its overhead is not transparent. We use a *recycled thread model* [21] to reduce the cost of thread initiation. Rather than create a new thread everytime prefetching is initiated, the prefetch thread is created once during program startup, and enters a blocking dispatch loop (Figure 18c). To initiate prefetching, the computation thread communicates a *PC* value through memory, and executes a `resume` instruction to dispatch the prefetch thread (Figure 18a, lines 3-4). After prefetching for the computation loop has been completed, the prefetch thread returns to the dispatch loop, thus "recycling" it for the next dispatch. In addition to thread initiation code, we also insert a `kill` instruction to terminate the prefetch thread in the event it is still active when the computation thread leaves the loop (Figure 18a, line 10).

Third, we insert code to pass arguments. Any variable used by the prefetch thread that is a local variable in the computation thread must be passed. Communication of arguments is performed through loads and stores to a special argument buffer in memory (Figure 18a, line 1 and Figure 18b, line 2). Although the computation thread's argument passing code is not executed transparently, we find this overhead is small since only a few arguments are typically passed and the argument buffer normally remains in cache.

Finally, we insert code to synchronize the prefetch thread with the computation thread. Because the prefetch thread executes only non-blocking memory references, it naturally gets ahead of the computation thread. We use a pair of loop-trip counters to keep the prefetch thread from getting too far ahead. One counter is updated by the computation thread (Figure 18a, line 6), and another is updated by the prefetch thread (Figure 18b, line 12). Every iteration, the prefetch thread compares the two counters, and continues only if they differ by less than the desired *prefetch distance* [15]; otherwise, the prefetch thread

(a) COMPUTATION THREAD

```
1   smt_global.param[0] = N;
2   producer = 0,  consumer  = 0;
3   resumeID = LOOP1;
4   resumeContext(cxt_id);
5     for (i=0; i<=N; i++) {
6       consumer++;
7        y = y + z[i];
8        x = x + a[b[i]];
9     }
10  KILL(cxt_id);
```

(c) DISPATCHER LOOP

```
1   void DISPATCHER() {
2     while(1) {
3       suspendContext(cxt_id);
4      (resumeID)();
5     }
6   }
```

(b) PREFETCH THREAD

```
1   void LOOP1() {
2    int N = smt_global.param[0];
3    /* Prologue */
4     for (i=0; i<PD; i++) {
5      prefetch(&b[i]);
6     }
7    /* Main Loop */
8     for (i=0; i<=N−PD; i++) {
9      prefetch(&b[i+PD]);
10     prefetch(&a[b[i]]);
11     prefetch(&z[i]);
12     producer++;
13       do {
14       } while (producer > consumer + PD);
15    }
16   /* Epilogue Loop */
17    for (; i<=N; i++) {
18     prefetch(&a[b[i]]);
19     prefetch(&z[i]);
20     producer++;
21       do {
22       } while (producer > consumer + PD);
23    }
```

Figure 18: TSP instrumentation example. (a) Computation thread code. (b) Prefetch thread code. (c) Dispatcher loop for implementing a recycled thread model.

busy-waits (Figure 18b, lines 13-14). While the prefetch thread may incur a significant number of busy-wait instructions, these instructions execute transparently.

Note, for indexed array references, we insert prologue and epilogue loops to software pipeline the index array and data array prefetches (Figure 18b, lines 3-6 and lines 16-23). This technique, borrowed from conventional software prefetching for indexed arrays [15], properly times the prefetch of serialized index array and data array references.

5.2 Performance Evaluation

In this section, we evaluate the performance of TSP, and compare it against two versions of conventional software prefetching: one that naively instruments prefetching for all load instructions, and one that uses detailed cache-miss profiles to instrument prefetching selectively. For selective software prefetching, we use the predicate in Equation 1 to evaluate prefetch profitability, and only instrument those static loads for which the predicate is true.

When evaluating the prefetch predicate from Equation 1, we assume an 8-cycle overhead per dynamic load (see Section 5.1). The L1 and L2 miss rates needed to evaluate the predicate are acquired by performing cache-miss profiling in each benchmark's simulation region given in Table 3, and we use the L2 hit time and memory latency values reported in Table 1. Once candidate loads have been selected using Equation 1, we instrument software prefetching by following the well-known algorithm in [15]. (We use the same algorithm to instrument software prefetching naively for all load instructions as well). In this section, instrumentation for both TSP and conventional software prefetching is performed by hand. Later in Section 5.3, we discuss preliminary work to automate TSP in a compiler.

Figure 19 presents performance results for the different prefetching schemes, using 7 out of the 8 benchmarks from Table 3 (we do not evaluate GZIP). In Figure 19, we report the normalized execution time for no prefetching (NP), naive software prefetching applied to all candidate loads (PF), selective software prefetching applied to loads meeting our predi-

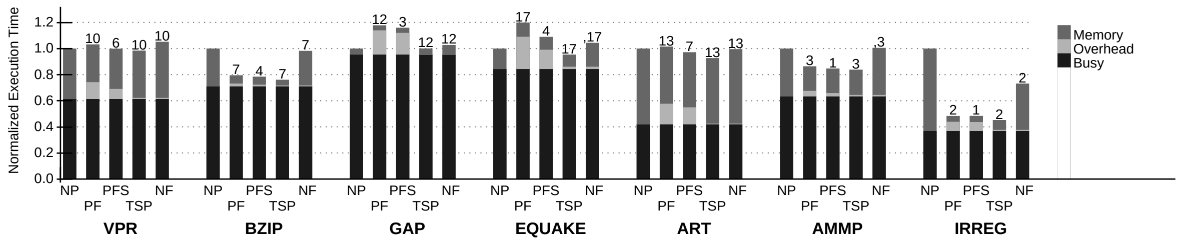

Figure 19: Normalized execution time for different prefetching schemes: no prefetching (NP), naive conventional software prefetching (PF), selective conventional software prefetching (PFS), Transparent Software Prefetching (TSP), and TSP without foreground thread flushing (NF). The label appearing above each bar reports the number of instrumented loops.

cate based on cache-miss profiles (PFS), and TSP applied to all candidate loads. Each bar in the graph is broken down into three components: time spent executing useful instructions, time spent executing prefetch-related instructions, and time spent stalled on data memory accesses, labeled "Busy," "Overhead," and "Memory," respectively. All values are normalized to the NP bars. Finally, a label appears above each bar reporting the number of instrumented loops. These numbers show a significant reduction in loop coverage when performing selective software prefetching.

Our results show TSP outperforms naive conventional software prefetching on every benchmark. Across the 6 SPEC benchmarks, TSP provides a 9.41% performance boost on average, whereas naive conventional software prefetching suffers a 1.38% performance degradation, reducing performance in 4 out of the 6 SPEC benchmarks. This performance discrepancy is due to a 19.6% overhead when using naive software prefetching compared to a 1.32% overhead when using TSP. Despite the fact that prefetching is instrumented for all candidate loads, TSP's negligible overhead enables it to avoid degrading performance even for overhead-sensitive benchmarks like GAP and EQUAKE where there is very little memory stall. Compared to naive software prefetching, selective software prefetching reduces overhead down to 14.13% by using profile information, resulting in a 2.47% performance gain averaged across the 6 SPEC benchmarks. However, TSP still outperforms selective software prefetching on every benchmark. Even for benchmarks where conventional software prefetching performs exceptionally well (*e.g.*, Irreg), TSP still performs better.

The performance gains demonstrated by TSP in Figure 19 suggest that transparent threads not only eliminate overhead, but they also provide enough resources for the prefetch threads to make sufficient forward progress. To evaluate the contribution of our transparent thread throughput mechanisms, the last set of bars in Figure 19, labeled "NF," report the normalized execution time of TSP without foreground thread flushing. The NF bars clearly show the complete set of mechanisms is critical since the performance gains of TSP are significantly reduced when foreground thread flushing is turned off.

5.3 Compiler Support

Sections 5.1 and 5.2 introduce TSP and evaluate its performance, showing good performance compared to conventional software prefetching even when applied naively. Another important issue is whether TSP can be automated in a compiler to relieve the programmer from the burden of manually generating code for the prefetch threads. Since conventional software prefetching has already been automated previously [15], it is important to show TSP can be similarly automated; otherwise, conventional software prefetching may still hold an advantage over TSP even if TSP achieves better performance.

To demonstrate TSP can be automated, we developed a compiler algorithm for extracting prefetch thread code via static analysis of an application's source code. This algorithm is presented in Appendix A. To show its feasibility, we prototyped the algorithm using the Stanford University Intermediate Format (SUIF) framework [22]. At the time of this paper's writing, we completed our compiler prototype and successfully generated prefetch thread code for two of our benchmarks, VPR and BZIP2, fully automatically. Furthermore, when executed on our simulator, the automatically generated TSP code performs identically compared to the manually generated code. Unfortunately, a complete study on automating TSP is beyond the scope of this paper. However, we believe our preliminary experience with compiler support suggests that automating TSP is feasible.

6. Related Work

SMT Processors [1, 23, 24] hold the state of multiple threads in hardware, allowing the execution of instructions from multiple threads each cycle on a wide superscalar processor. This organization has the potential to boost the throughput of the processor. The priority that different threads receive in utilizing the processor resources is determined by resource allocation policies.

Several researchers have studied hardware resource allocation mechanisms [2, 23, 25] and operating system scheduling policies [26, 27] for SMT processors. In particular, Tullsen and Brown [25] first proposed flushing to reclaim execution resources stalled on long latency memory operations. Their work was the motivation behind several of our mechanisms. Compared to these previous techniques, however, our work tries to improve single-thread performance whereas the previous work focuses solely on processor throughput.

Raasch and Reinhardt [28] proposed fetch policies for SMT processors that consider single-thread performance in addition to overall processor throughput. They assume a single latency-critical foreground thread executes simultaneously with one or more low-priority background threads, and evaluate fetch policies that favor the foreground thread over the background thread(s). Luo et al [2] also discuss the notion of fairness in SMT processors so that less efficient threads do not get starved of processor resources. Our work focuses on mechanisms that permit background threads to share resources with the foreground thread in a completely transparent fashion. Furthermore, we apply priority mechanisms for slots and buffers along the entire pipeline, rather than just for the fetch stage.

Software prefetching [17, 18, 15] is a promising technique to mitigate the memory latency bottleneck. Traditional software prefetching techniques inline the prefetch code along with the main computation code. We propose Transparent Software Prefetching, a novel subordinate threading technique that executes prefetch code in transparent threads. Chappell *et al* [3] and Dubois and Song [4] proposed subordinate threads as a means for improving main thread performance. In [4], the authors demonstrate stride prefetching can be implemented in software using subordinate threads. Our TSP technique is similar, but we use transparent threading mechanisms to eliminate the overhead of the subordinate prefetch threads.

Subordinate threads have also been used to execute exception handlers [29], and to pre-execute performance-degrading instructions [5, 6, 7, 8, 9, 10]. These latency tolerance techniques use one or more helper threads running ahead of the main computation to trigger long-latency memory operations early. They can also assist branch prediction by resolving hard-to-predict branches early [10]. Our work could be used to minimize the overhead of these techniques as well.

7. Conclusions

This paper investigates resource allocation mechanisms for SMT processors that preserve, as much as possible, the single-thread performance of designated foreground threads, while still allowing background or "transparent" threads to share resources. Our mechanisms ensure transparent threads never take performance-critical resources away from the foreground thread, yet aggressively allocate those resources to transparent threads that do not contribute to foreground thread performance. The conclusions of our work can be enumerated as follows.

Transparency mechanisms. Instruction buffer occupancy is an important parameter that determines the impact of background threads on the foreground thread. In order to guarantee at allocation time that buffer resources can be given to background threads transparently, we would need future information as to how long a background thread's instructions will occupy an instruction buffer. Since such information cannot be obtained in a realistic machine, we use pre-emption to reclaim buffer entries from background threads whenever they are needed by the foreground thread. On a suite of multiprogramming workloads, our results show transparent threads introduce a 3% foreground thread performance degradation on average, and when contention on cache and branch predictor resources are factored out, the performance degradation is less than 1% for all workloads.

Performance mechanisms. Foreground threads with long latency operations tend to overwhelm the instruction buffer resources, thereby adversely impacting background thread performance. This work shows the importance of aggressively allocating stagnant foreground thread buffer resources to the background thread in order to boost its overall throughput. Our results show that transparent threads run only 23% slower compared to an equal priority scheme when using our foreground thread flushing mechanism.

Transparent Software Prefetching. To demonstrate the potential uses of transparent threads, our work also proposes an implementation of software prefetching on transparent threads, called Transparent Software Prefetching. TSP solves the classic problem of determining what to prefetch. Due to the near-zero overhead of transparent threads, TSP can be applied naively, without ever worrying about the profitability of a transformation. In our evaluation of Transparent Software Prefetching, our results show TSP achieves a 9.41% performance gain across 6 SPEC benchmarks, whereas conventional software prefetching degrades performance by 1.38%. Even when detailed cache-miss profiles are used to guide instrumentation selectively, conventional software prefetching only achieves a 2.47% performance gain. The performance advantage of TSP comes from its 1.32% overhead, compared to a 14.13% overhead for selective software prefetching.

Spare Execution Bandwidth. Based on our preliminary results, we conclude that applications running on wide out-of-order superscalar cores leave a significant number of unused resources that can be allocated to non-critical computations in a completely non-intrusive fashion. We believe our work has only begun to look at the potential uses for such "free" execution bandwidth. In future work, we hope to further explore the applications of transparent threads, including multiprogrammed workload scheduling, subordinate multithreading optimization, and on-line performance monitoring, as eluded to at the beginning of this paper.

8. Acknowledgments

The authors would like to thank Dongkeun Kim for contributing to the simulator development effort. We would also like to thank the anonymous referees for their constructive comments on earlier drafts of this paper.

Appendix A. Compiler Algorithm for Instrumenting TSP

Figure 20 presents our algorithms for automatically generating prefetch thread code for TSP. The first algorithm, `Prefetch_Thread()` in Figure 20a, extracts the following information: the set of loop induction variables used as array reference indices, LI, the set of loop exit conditions, EC, the set of memory references for prefetching, LD, and the set of parameters passed from the main thread to the prefetch thread, PR. The `Prefetch_Thread()` algorithm is applied to each inner-most loop in the application source code. Together, LI, EC, LD, and PR are sufficient to generate the prefetch thread code. (The code generation steps are not shown in Figure 20; in our SUIF compiler prototype, we use a Perl script to generate prefetch thread code once LI, EC, LD, and PR have been extracted by SUIF).

The first step is to compute LI. A statement S is said to be a *loop induction statement* if S both uses (*USE(S)*) and defines (*KILL(S)*) a variable, and the definition reaches the next invocation of S in the following iteration (*REACH(S)*) without being overwritten by another definition. Lines 3-9 compute LI accordingly. The second step is to compute EC to enable proper termination of the prefetch thread. Specifically, the prefetch thread should evaluate all conditional expressions from the main thread code that control loop termination, as well as all statements affecting such loop termination conditions. To include all relevant code, we perform *backward slicing* [30] on the conditional expressions using the `Backslice()` algorithm (explained below). Lines 10-14 compute EC. The third step is to find the set of memory references for prefetching, LD. We prefetch all affine array (of the form A[i]) or indexed array (of the form A[B[i]]) references. Similar to computing exit conditions, we rely on backward slicing to extract all relevant code for computing prefetch memory addresses. Lines 15-21 compute LD.

While computing LI, EC, and LD, our algorithm in Figure 20a also determines the live-in variables to the prefetch thread code. All local variables that are used before their first define inside the prefetch loop should be passed to the prefetch thread as parameters. Each

```
(a) Given:
        Inner most loop, LP
    Computes:
        Set of parameters for prefetch thread, PR
        Set of loop induction statements, LI
        Set of loop exit conditions, EC
        Set of prefetchable loads, LD
    ________________________________________________
 1  Prefetch_Thread(LP) {
 2    PR = LI = EC = LD = $\Phi$
 3    do {
 4      if (S in KILL(S) && S in REACH(S) && S in USE(S)) {
 5        <_param, _dummy, _dummy> = Backslice(S, S, LP, LI, $\Phi$)
 6        PR = PR U _param
 7        LI = LI U S
 8      }
 9    } $\forall$ variable defining statements S in LP
10    do {
11      <_param, _slice, _dummy> = Backslice(S, S, LP, LI, $\Phi$)
12      PR = PR U _param
13      EC = EC U <S, _slice>
14    } $\forall$ loop exit conditions S in LP
15    do {
16      <_param, _slice, _index> = Backslice(S, S, LP, LI, LD)
17      if (_index != $\Phi$ && backslice not aborted)
18        PR = PR U _param
19        LD = LD U <S, _slice, _index>
20      }
21    } $\forall$ memory references S in LP
22    do {
23      if (S accesses the same cache block as one in LD)
24        LD = LD - S
25    } $\forall$ memory references S in LD {
26    return <PR, LI, EC, LD>
27  }
```

```
(b) Given:
        Memory reference statement, SM
        Current statement being inspected for backward slice, SC
        Target loop, LP
        Set of loop induction statements, LI
        Set of prefetchable loads, LD
    Computes:
        Parameter list, PR
        Backward slices for SM's address calculation, SL
        Index of SM, IX
    ________________________________________________
28  Backslice(SM, SC, LP, LI, LD) {
29    PR = SL = IX = $\Phi$
30    do {
31      _param = _slice = _index = $\Phi$
32      R = {D | D in REACH(SC) and D defines V}
33      if (|R| > 1)
34        abort backslicing
35      else if (R  == SM)
36        abort backslicing
37      else if (R == $\Phi$)
38        _param = V
39      else if (R in LI)
40        _index = R
41      else if (R in LD)
42        _index = R
43      else
44        <_param, _slice, _index> = Backslice(SM, R, LP, LI, LD)
45      PR = PR U _param
46      SL = SL U R U _slice
47      IX = IX U _index
48    } $\forall$ input variables V in SC
49    return <PR, SL, IX>
50  }
```

Figure 20: Algorithm for generating prefetch thread code automatically. (a) `Prefetch_Thread()` computes all information necessary to generate prefetch thread code. (b) `Backslice()` performs backward slicing to extract code statements for computing exit conditions and prefetch memory addresses.

live-in is identified during backward slicing analysis, and is then collected into the set of parameters, PR, at lines 6, 12, and 18. Finally, locality analysis is performed in lines 22-25 to prevent multiple memory accesses to the same cache block, thus avoiding redundant prefetches.

The second algorithm, `Backslice()` in Figure 20b, computes information needed by the `Prefetch_Thread()` algorithm. First, `Backslice()` computes the backward slice, SL, which is the chain of dependent statements leading up to a specified statement, SM. Backward slicing, as mentioned earlier, extracts the relevant code for exit conditions and prefetch memory addresses. In addition, `Backslice()` also computes the set of index variable statements, IX, in each backward slice associated with prefetchable load instructions. Finally, `Backslice()` also identifies all live-in variables, PR, in each backward slice.

`Backslice()` recursively traverses the control flow graph of a loop, LP, starting from SM, and identifies all reaching definitions for the input variables at statement SC (the current point of analysis). Termination of recursion can occur in one of three ways. First, when no statement in the loop defines an input variable V at statement SC, the variable is loop invariant (lines 37-38). Such loop-invariant variables are added to the parameter set, PR, signifying their values must be passed into the prefetch thread code as parameters. Second, if the reaching definition is itself a loop induction statement (*i.e.*, it is a member of set LI) and SM is a memory reference (which occurs when `Backslice()` is called from line 16), then SM contains an affine array reference (lines 39-40). Third, if the reaching definition involves an affine array or indexed array reference (*i.e.*, it is a member of set LD) and SM is a memory reference, then SM contains an indexed array reference (lines 41-42). For both affine and indexed array references, we include the reaching definition in the set of index variable statements, IX, so that the index can be identified for each array reference (this information is used in line 19). If SM is a memory reference but recursion does not terminate at lines 39-42, then SM is loop invariant and thus accesses a scalar variable. In this case, we will return a null index set, signifying that this memory reference should not be prefetched (line 17).

In addition to the three conditions described above that terminate recursion, there are two error conditions that can cause the entire slicing analysis to abort. Our algorithm does not prefetch a load whose address is computed along multiple paths. Hence, `Backslice()` aborts when it detects a variable has multiple reaching definitions (lines 33-34). In addition, our algorithm does not prefetch pointer-chasing loads. This restriction causes `Backslice()` to abort when a variable is defined as a function of itself (lines 35-36). If none of the terminating or aborting conditions apply, then our algorithm recurses to continue slicing (lines 43-44).

References

[1] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," in *Proceedings of the 1996 International Symposium on Computer Architecture*, (Philadelphia), May 1996.

[2] K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec, "Boosting SMT Performance by Speculation Control," in *Proceedings of the International Parallel and Distributed Processing Symposium*, (San Francisco, CA), April 2001.

[3] R. S. Chappell, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous Subordinate Microthreading (SSMT)," in *In Proceedings of the 26th International Symposium on Computer Architecture*, (Atlanta, GA), pp. 186–195, ACM, May 1999.

[4] M. Dubois and Y. H. Song, "Assisted Execution." October 1998.

[5] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data Prefetching by Dependence Graph Precomputation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, (Goteborg, Sweden), ACM, June 2001.

[6] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic Speculative Precomputation," in *Proceedings of the 34th International Symposium on Microarchitecture*, (Austin, TX), December 2001.

[7] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, “Speculative Precomputation: Long-range Prefetching of Delinquent Loads,” in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, (Goteborg, Sweden), June 2001.

[8] C.-K. Luk, “Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,” in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, (Goteborg, Sweden), ACM, June 2001.

[9] A. Roth and G. S. Sohi, “Speculative Data-Driven Multithreading,” in *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pp. 191–202, January 2001.

[10] C. Zilles and G. Sohi, “Execution-Based Prediction Using Speculative Slices,” in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, (Goteborg, Sweden), June 2001.

[11] E. G. Hallnor and S. K. Reinhardt, “A Fully Associative Software-Managed Cache Design,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (Vancouver, Canada), June 2000.

[12] T. Ball and J. R. Larus, “Efficient Path Profiling,” in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, (Paris, France), IEEE, December 1996.

[13] J. R. Larus and eric Schnarr, “EEL: Machine-Independent Executable Editing,” in *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation*, June 1995.

[14] A. Srivastava and alan Eustace, “ATOM: A System for Building Customized Program Analysis Tools,” in *Proceedings of the SIGPLAN ’94 Conference on Programming Language Design and Implementation*, pp. 196–205, June 1994.

[15] T. Mowry, “Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching,” *Transactions on Computer Systems*, vol. 16, pp. 55–92, February 1998.

[16] D. Madon, E. Sanchez, and S. Monnier, “A Study of a Simultaneous Multithreaded Processor Implementation,” in *Proceedings of EuroPar ’99*, (Toulouse, France), pp. 716–726, Springer-Verlag, August 1999.

[17] D. Callahan, K. Kennedy, and A. Porterfield, “Software Prefetching,” in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 40–52, April 1991.

[18] A. C. Klaiber and H. M. Levy, “An Architecture for Software-Controlled Data Prefetching,” in *Proceedings of the 18th International Symposium on Computer Architecture*, (Toronto, Canada), pp. 43–53, ACM, May 1991.

[19] S. G. Abraham, R. A. Sugumar, B. R. Rau, and R. Gupta, "Predictability of Load/Store Instruction Latencies," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pp. 139–152, IEEE/ACM, 1993.

[20] T. C. Mowry and C.-K. Luk, "Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, IEEE/ACM, December 1997.

[21] J. G. Steffan, C. B. Colohan, and T. C. Mowry, "Architectural Support for Thread-Level Data Speculation," CMU-CS 97-188, Carnegie Mellon University, November 1997.

[22] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, vol. 29, Dec. 1996.

[23] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 392–403, ACM, June 1995.

[24] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," *IEEE Micro*, pp. 12–18, September/October 1997.

[25] D. M. Tullsen and J. A. Brown, "Handling Long-latency Loads in a Simultaneous Multithreading Processor," in *Proceedings of the 34th International Symposium on Microarchitecture*, (Austin, TX), December 2001.

[26] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, (Cambridge, MA), November 2000.

[27] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2002.

[28] S. E. Raasch and S. K. Reinhardt, "Applications of Thread Prioritization in SMT Processors," in *Proceedings of the 1999 Multithreaded Execution, Architecture, and Compilation Workshop*, January 1999.

[29] C. B. Zilles, J. S. Emer, and G. S. Sohi, "The Use of Multithreading for Exception Handling," in *Proceedings of the 32nd International Symposium on Microarchitecture*, pp. 219–229, November 1999.

[30] C. B. Zilles and G. S. Sohi, "Understanding the Backward Slices of Performance Degrading Instructions," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (Vancouver, Canada), pp. 172–181, June 2000.